Hardware Verification Project Report

Group member: Hang Ye (hy2891), Junfeng Zou(jz3850), Yunyang Lu(yl5764)

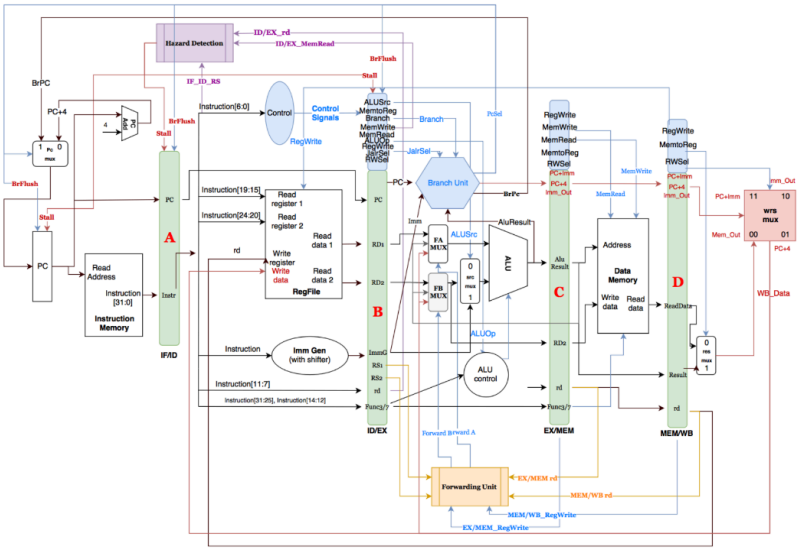
Introduction

The RISC(Reduced Instruction Set Computing)-V processor is an open-source, modular instruction set architecture which is widely researched and implemented in various applications. Despite its popularity, the formal verification of RISC-V designs are not always available. This project focuses on verifying key functionalities of a RISC-V processor's pipeline architecture implemented in SystemVerilog, which is available in https://github.com/estufa-cin-ufpe/RISC-V-Pipeline. The RISC-V processor features a multi-stage pipeline and follows the principles of Reduced Instruction Set Computing, designed to execute a simplified set of instructions efficiently. The project employs JasperGold to formally verify certain properties in these stages, including the register files, the PCs which spread through each stage, the stall and the scratch signal.

Background

The RISC-V processor is an open-source, modular instruction set architecture (ISA) based on the RISC (Reduced Instruction Set Computer) principles, designed for flexibility, scalability, and efficiency. Unlike proprietary ISAs like ARM or x86, RISC-V is free to use and modify, making it popular in research, academia, and industry. Its simplicity, extensibility, and support for multiple instruction widths (32, 64, and 128 bits) allow it to fit a wide range of applications, from low-power IoT devices to high-performance computing systems. Its open nature has fostered a growing ecosystem, accelerating adoption and innovation in various fields.

This pipeline diagram below illustrates the five stages of the RISC-V processor—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage processes different parts of the instruction, from fetching it from memory to performing operations in the ALU and accessing memory, ending with writing the result back to the register file. Key components like the hazard detection and forwarding units manage data hazards and control the data flow to ensure efficient and accurate instruction execution, enabling pipelined processing for faster performance.



Verification Goal

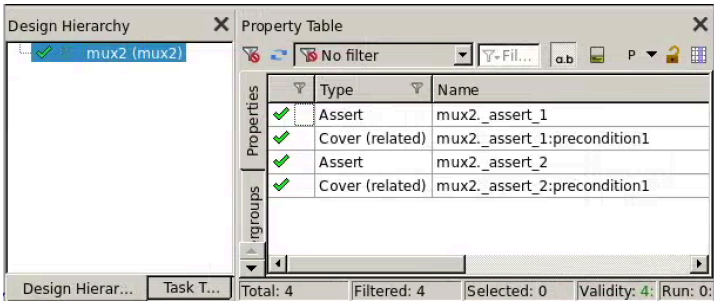
We aim to verify part of the processor modules, i.e. mux, register, alu, etc. The verification will primarily use the Jasper Gold tool. The verification will use a property based approach, instead of simulation based, to try to perform an exhaustive verification.

For verification of the mux file, (mux2.sv)

1. Safety Property
    a. When s is 0, y should be equal to d0.
    b. When s is 1, y should be equal to d1.
2. Concurrent Assertions

```
assert property (
    (s == 1'b0) |-> (y == d0)
) else $error("When s is 0, y should be equal to d0");

assert property (
    (s == 1'b1) |-> (y == d1)
) else $error("When s is 1, y should be equal to d1");
endmodule
```

3. Property Proof



For verification of the register file, (Regfile.sv)

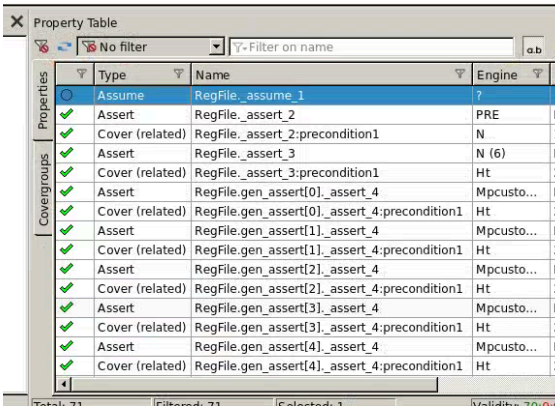
1. Properties

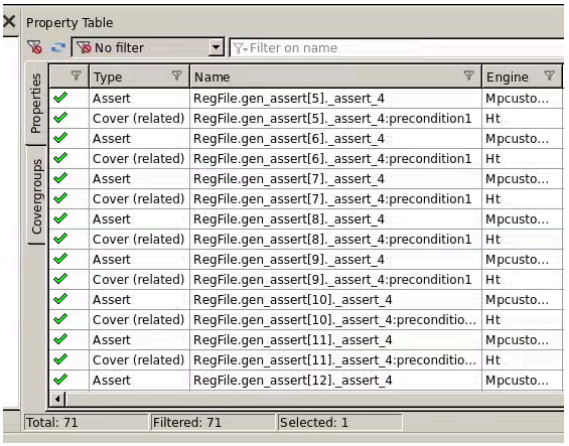
- Safety Property
    a. When there is a write request, the registers in other addresses should not change their value.
    b. The values stored in the register file are cleared to 0 after the reset signal is received.
    c. A change in read register address(es) should switch the correct corresponding Read data output in one clock cycle.
- Liveness Property
    a. If there is a write request, the register in the designated address will eventually update to the write data value (or be updated to the correct value in expected cycles).

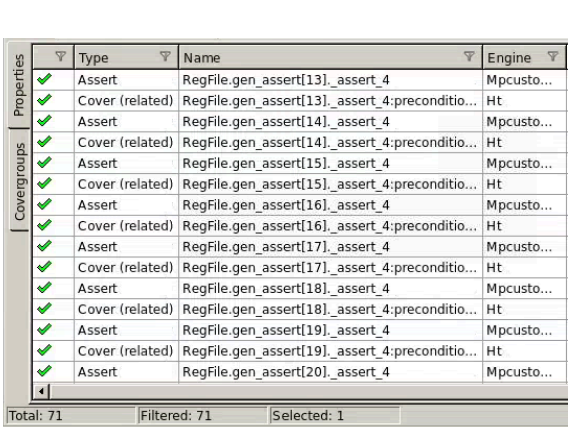
2. Assertions

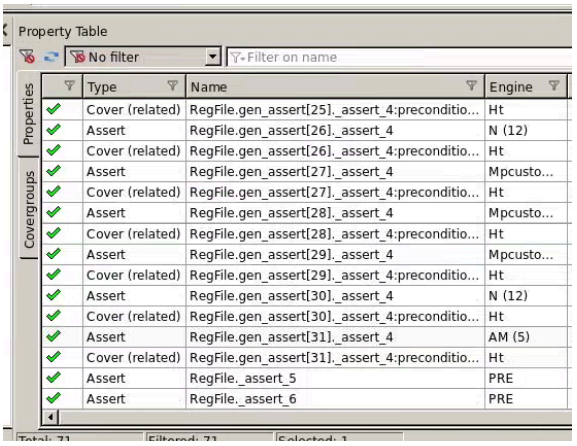
```
40
41 assume property($stable(rg_wrt_dest) && $stable(rg_wrt_en));
42
43 assert property (@(negedge clk)
44     $past(rst) |-> (register_file == '{default: 0})
45 ) else $error("All registers should be reset to 0 during reset");
46
47 assert property (@(negedge clk)
48     $past(rg_wrt_en && !rst) |-> (register_file[rg_wrt_dest] == $past(rg_wrt_data))
49 ) else $error("Data written should match rg_wrt_data during write operation");
50
51 genvar j;
52 generate
53     for (j = 0; j < NUM_REGS; j = j + 1) begin : gen_assert
54         assert property (@(negedge clk)
55             $past(rg_wrt_en && !rst) |->
56             (register_file[j] == $past(register_file[j]) || (j == rg_wrt_dest))
57         ) else $error("Data written should not change other entries");
58     end
59 endgenerate
60
61 assert property (@(negedge clk) disable iff (rst)
62     (rg_rd_data1 == register_file[rg_rd_addr1])
63 ) else $error("Read data1 should match the data in register_file at rg_rd_addr1");
64
65 assert property (@(negedge clk) disable iff (rst)
66     (rg_rd_data2 == register_file[rg_rd_addr2])
67 ) else $error("Read data2 should match the data in register_file at rg_rd_addr2");
68 endmodule
```

3. Property Proof









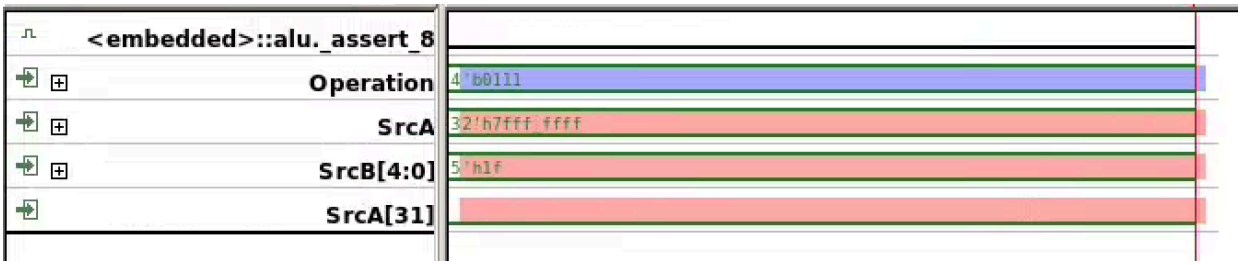
For verification on ALU, (alu.sv)

1. Safety Assertions
   Each function works correctly (and, or, add, xor, shift left, shift right, subtract, right shift arithmetic, equal, not equal, less than, greater/equal than, unsigned less than, unsigned greater/equal than, jal, default).
2. Concurrent Assertions
   Only the right shift arithmetic function failed in the verification.

```
4'b0111:        // Arithmetic Right Shift
        ALUResult = $signed(SrcA) >>> SrcB[4:0];
```

```
assert property (  (Operation == 4'b0111) |-> ($signed(SrcA) >>> SrcB[4:0]) //Arithmetic Right Shift
) else $error("ALU Arithmetic Right Shift operation failed: ALUResult does not match $signed(SrcA) >>> SrcB[4:0]");
```

✗	Assert	alu._assert_8	Hp	1	1
✓	Cover (related)	alu._assert_8:precondition1	Hp	1	1



The violation occurred because the arithmetic right shift operation did not produce the expected result. Specifically, with "SrcA = 32'h7FFFFFFF" (the largest positive 32-bit signed integer) and "SrcB = 5'h1F" (shift amount of 31), the expected result should be 0, as shifting the value 31 times preserves only the sign bit (0 for positive numbers). The ALU might be incorrectly implementing logical right shift ">>" instead of arithmetic right shift ">>>", which does not preserve the sign bit. The property may also not correctly capture the expected behavior due to a mismatch in how the assertion or simulation interprets the signals, or the input "SrcA" may not have been correctly treated as a signed value.
Finally, we modified the arithmetic right shift function to inversion function.

```
assert property (  (Operation == 4'b0000) |-> (ALUResult == (SrcA & SrcB)) //AND
)else $error("ALU AND operation failed: ALUResult does not match SrcA & SrcB");

assert property (  (Operation == 4'b0001) |-> (ALUResult == (SrcA | SrcB)) //OR
) else $error("ALU OR operation failed: ALUResult does not match SrcA | SrcB");

assert property (  (Operation == 4'b0010) |-> (ALUResult == ($signed(SrcA) + $signed(SrcB))) //ADD
) else $error("ALU ADD operation failed: ALUResult does not match SrcA + SrcB");

assert property (  (Operation == 4'b0011) |-> (ALUResult == (SrcA ^ SrcB)) //XOR
) else $error("ALU XOR operation failed: ALUResult does not match SrcA ^ SrcB");

assert property (  (Operation == 4'b0100) |-> (ALUResult == (SrcA << SrcB[4:0])) //Left Shift
) else $error("ALU Left Shift operation failed: ALUResult does not match SrcA << SrcB[4:0]");

assert property (  (Operation == 4'b0101) |-> (ALUResult == (SrcA >> SrcB[4:0])) //Right Shift
) else $error("ALU Right Shift operation failed: ALUResult does not match SrcA >> SrcB[4:0]");

assert property (  (Operation == 4'b0110) |-> (ALUResult == ($signed(SrcA) - $signed(SrcB))) //Subtract
) else $error("ALU Subtract operation failed: ALUResult does not match SrcA - SrcB");

assert property (  (Operation == 4'b0111) |-> (ALUResult == ~SrcA) //Invert A
) else $error("ALU Inversion operation failed: ALUResult does not match ~SrcA");
```

```
assert property (  (Operation == 4'b1000) |-> (ALUResult == ((SrcA == SrcB) ? 1 : 0)) //Equal
) else $error("ALU Equal operation failed: ALUResult does not match (SrcA == SrcB) ? 1 : 0");

assert property (  (Operation == 4'b1001) |-> (ALUResult == ((SrcA !== SrcB) ? 1 : 0)) //Not Equal
) else $error("ALU Not Equal operation failed: ALUResult does not match (SrcA !== SrcB) ? 1 : 0");

assert property (  (Operation == 4'b1100) |-> (ALUResult == (($signed(SrcA) < $signed(SrcB)) ? 1 : 0)) //Less Than
) else $error("ALU Less Than operation failed: ALUResult does not match ($signed(SrcA) < $signed(SrcB)) ? 1 : 0");

assert property (  (Operation == 4'b1101) |-> (ALUResult == (($signed(SrcA) >= $signed(SrcB)) ? 1 : 0)) //Greater Than/Equal To
) else $error("ALU Greater/Equal Than operation failed: ALUResult does not match ($signed(SrcA) >= $signed(SrcB)) ? 1 : 0");

assert property (  (Operation == 4'b1110) |-> (ALUResult == ((SrcA < SrcB) ? 1 : 0)) //Unsigned Less Than
) else $error("ALU Unsigned Less Than operation failed: ALUResult does not match (SrcA < SrcB) ? 1 : 0");

assert property (  (Operation == 4'b1111) |-> (ALUResult == ((SrcA >= SrcB) ? 1 : 0)) //Unsigned Greater Than/Equal To
) else $error("ALU Unsigned Greater/Equal Than operation failed: ALUResult does not match (SrcA >= SrcB) ? 1 : 0");

assert property (  (Operation == 4'b1010) |-> (ALUResult == 1) //jal
) else $error("ALU jal operation failed: ALUResult does not match 1");

assert property (  (Operation == 4'b1011) |-> (ALUResult == 0) //default
) else $error("ALU default operation failed: ALUResult does not match 0");
```

3. Property Proof

✔	Assert	alu._assert_1	N (1)	Infinite	0
✔	Cover (related)	alu._assert_1:precondition1	N	1	1
✔	Assert	alu._assert_2	Hp (1)	Infinite	0
✔	Cover (related)	alu._assert_2:precondition1	N	1	1
✔	Assert	alu._assert_3	Hp (1)	Infinite	0
✔	Cover (related)	alu._assert_3:precondition1	N	1	1
✔	Assert	alu._assert_4	Hp (1)	Infinite	0
✔	Cover (related)	alu._assert_4:precondition1	N	1	1
✔	Assert	alu._assert_5	Hp (1)	Infinite	0
✔	Cover (related)	alu._assert_5:precondition1	N	1	1
✔	Assert	alu._assert_6	Hp (1)	Infinite	0
✔	Cover (related)	alu._assert_6:precondition1	N	1	1
✔	Assert	alu._assert_7	Hp (1)	Infinite	0
✔	Cover (related)	alu._assert_7:precondition1	Hp	1	1
✔	Assert	alu._assert_8	Hp (1)	Infinite	0

✔	Assert	alu._assert_9	Hp (1)	Infinite
✔	Cover (related)	alu._assert_9:precondition1	Hp	1
✔	Assert	alu._assert_10	Hp (1)	Infinite
✔	Cover (related)	alu._assert_10:precondition1	Hp	1
✔	Assert	alu._assert_11	Hp (1)	Infinite
✔	Cover (related)	alu._assert_11:precondition1	Hp	1
✔	Assert	alu._assert_12	Hp (1)	Infinite
✔	Cover (related)	alu._assert_12:precondition1	Hp	1
✔	Assert	alu._assert_13	Hp (1)	Infinite
✔	Cover (related)	alu._assert_13:precondition1	Hp	1
✔	Assert	alu._assert_14	Hp (1)	Infinite
✔	Cover (related)	alu._assert_14:precondition1	PRE	1
✔	Assert	alu._assert_15	Hp (1)	Infinite
✔	Cover (related)	alu._assert_15:precondition1	Hp	1
✔	Assert	alu._assert_16	Hp (1)	Infinite
✔	Cover (related)	alu._assert_16:precondition1	Hp	1

For Controller(controller.sv)

```
module Controller(

    //Input
    input logic [6:0] Opcode, //7-bit opcode field from the instruction

    //Outputs
    output logic ALUSrc,//0: The second ALU operand comes from the second register file output (Read data 2);
                        //1: The second ALU operand is the sign-extended, lower 16 bits of the instruction.
    output logic MemtoReg, //0: The value fed to the register Write data input comes from the ALU.
                           //1: The value fed to the register Write data input comes from the data memory.
    output logic RegWrite, //The register on the Write register input is written with the value on the Write data input
    output logic MemRead,  //Data memory contents designated by the address input are put on the Read data output
    output logic MemWrite, //Data memory contents designated by the address input are replaced by the value on the Write data input.
    output logic [1:0] ALUOp,   //00: LW/SW/AUIPC; 01:Branch; 10: Rtype/Itype; 11:JAL/LUI
    output logic Branch,  //0: branch is not taken; 1: branch is taken
    output logic JalrSel,      //0: Jalr is not taken; 1: jalr is taken
    output logic [1:0] RWSel    //00: Register Write Back; 01: PC+4 write back(JAL/JALR); 10: imm-gen write back(LUI); 11: pc+imm-gen write back(AUIPC)
);
```

1. Features

Features can be analyzed from the input ports and output ports.

ALUSrc: Controls the source of the second ALU operand.
0: From the second register file output.
1: From the immediate field of the instruction.
MemtoReg: Controls the source of data to be written back to the register.
0: From the ALU result.
1: From the data memory.
RegWrite: Controls whether to write to the register file.
0: Do not write to the register file.
1: Write to the register file.
MemRead: Controls whether to read from the data memory.
0: Do not read from the data memory.
1: Read from the data memory.
MemWrite: Controls whether to write to the data memory.
0: Do not write to the data memory.
1: Write to the data memory.
ALUOp: Controls the type of ALU operation.
00: LW/SW/AUIPC.
01: Branch.
10: Rtype/Itype.
11: JAL/LUI.
Branch: Controls whether to take a branch.
0: Do not take a branch.
1: Take a branch.
JalrSel: Controls whether to perform a JALR operation.
0: Do not perform a JALR operation.
1: Perform a JALR operation.
RWSel: Controls the source of data to be written back to the register.
00: Register write back.
01: PC+4 write back (JAL/JALR).
10: Immediate generation write back (LUI).
11: PC+immediate generation write back (AUIPC).

2. Property and assertions

Due to the reason that all the property is assigned by a combinational logic circuit, and should be changed immediately, the property can be seen as a safety property. The property assertion can be written as below:

```
assert property (
    (Opcode == LW) |-> (ALUSrc == 1 && MemtoReg == 1 && RegWrite == 1 && MemRead == 1 && MemWrite == 0 && ALUOp == 2'b00 && Branch == 0 && JalrSel == 0 && RWSel == 2'b00)
) else $error("LW instruction failed");

assert property (
    (Opcode == SW) |-> (ALUSrc == 1 && MemtoReg == 0 && RegWrite == 0 && MemRead == 0 && MemWrite == 1 && ALUOp == 2'b00 && Branch == 0 && JalrSel == 0 && RWSel == 2'b00)
) else $error("SW instruction failed");

assert property (
    (Opcode == R_TYPE) |-> (ALUSrc == 0 && MemtoReg == 0 && RegWrite == 1 && MemRead == 0 && MemWrite == 0 && ALUOp == 2'b10 && Branch == 0 && JalrSel == 0 && RWSel == 2'b00)
) else $error("R_TYPE instruction failed");

assert property (
    (Opcode == BR) |-> (ALUSrc == 0 && MemtoReg == 0 && RegWrite == 0 && MemRead == 0 && MemWrite == 0 && ALUOp == 2'b01 && Branch == 1 && JalrSel == 0 && RWSel == 2'b00)
) else $error("BR instruction failed");

assert property (
    (Opcode == JAL) |-> (ALUSrc == 0 && MemtoReg == 0 && RegWrite == 1 && MemRead == 0 && MemWrite == 0 && ALUOp == 2'b11 && Branch == 1 && JalrSel == 0 && RWSel == 2'b01)
) else $error("JAL instruction failed");

assert property (
    (Opcode == JALR) |-> (ALUSrc == 1 && MemtoReg == 0 && RegWrite == 1 && MemRead == 0 && MemWrite == 0 && ALUOp == 2'b00 && Branch == 0 && JalrSel == 1 && RWSel == 2'b01)
) else $error("JALR instruction failed");

assert property (
    (Opcode == LUI) |-> (ALUSrc == 0 && MemtoReg == 0 && RegWrite == 1 && MemRead == 0 && MemWrite == 0 && ALUOp == 2'b11 && Branch == 0 && JalrSel == 0 && RWSel == 2'b10)
) else $error("LUI instruction failed");

assert property (
    (Opcode == AUIPC) |-> (ALUSrc == 0 && MemtoReg == 0 && RegWrite == 1 && MemRead == 0 && MemWrite == 0 && ALUOp == 2'b00 && Branch == 0 && JalrSel == 0 && RWSel == 2'b11)
) else $error("AUIPC instruction failed");

// Assertion to check that Opcode does not match any of the above cases
assert property (
    !(Opcode == LW || Opcode == SW || Opcode == R_TYPE || Opcode == BR || Opcode == JAL || Opcode == JALR || Opcode == LUI || Opcode == AUIPC)
) else $error("Invalid Opcode detected");
```

The first 8 property assertions are written by following each feature. The last property assertion is to make sure the input can only be one of the eight cases, in order to prevent the unpredictable results.

3. Results

Properties | Covergroups

	Type	Name	Engine	Bound	Traces	Time	Task
✓	Assert	Controller._assert_1	N (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_1:precondition1	Hp	1	1	0.2	<embedde
✓	Assert	Controller._assert_2	Hp (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_2:precondition1	Hp	1	1	0.2	<embedde
✓	Assert	Controller._assert_3	Hp (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_3:precondition1	Hp	1	1	0.3	<embedde
✓	Assert	Controller._assert_4	Hp (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_4:precondition1	Hp	1	1	0.3	<embedde
✓	Assert	Controller._assert_5	Hp (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_5:precondition1	Hp	1	1	0.3	<embedde
✓	Assert	Controller._assert_6	Hp (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_6:precondition1	Hp	1	1	0.4	<embedde
✓	Assert	Controller._assert_7	Hp (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_7:precondition1	Hp	1	1	0.4	<embedde
✓	Assert	Controller._assert_8	Hp (1)	Infinite	0	<0.1	<embedde
✓	Cover (related)	Controller._assert_8:precondition1	Hp	1	1	0.4	<embedde
✗	Assert	Controller._assert_9	Hp	1	1	0.2	<embedde

Result shows that the last assertion failed. That is because Opcode is an input from the uplink module. There are implicit limits on it, while jasper would only generate random input to test.

4. How to run the JG for this component.

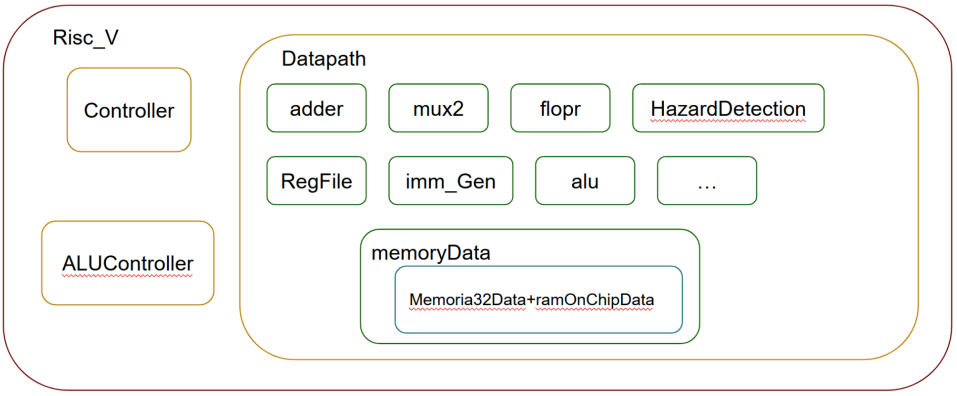
under the folder “controller”, type the following command in the terminal:

./run.sh

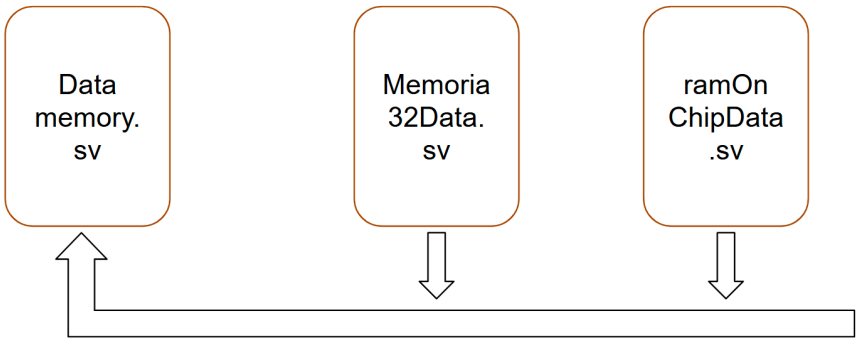
For hierarchy design

In the last session, the last assertion on the inputs fails because of the implicit limitation from uplink design, which can be seen as hierarchy design. Here is a picture for the design we have in a hierarchical view.

```
[hy2891@cadpc07 design]$ ls *v
adder.sv          Datapath.sv          Memoria32Data.sv  RegFile.sv
ALUController.sv  flopr.sv             Memoria32.sv      RegPack.sv
alu.sv            ForwardingUnit.sv    mux2.sv           RISC_V.sv
BranchUnit.sv     HazardDetection.sv   mux4.sv
Controller.sv     imm_Gen.sv           ramOnChip32.v
datamemory.sv     instructionmemory.sv ramOnChipData.v
```



So, we tried to do the verification on memoryData.sv for a hierarchy verification.



From the source code, we can get the hierarchy diagram above. We checked all three files, and found that there is an IP block at the file “ramOnChipData.sv”. It is Primitive-level IP core called Altera Synchronous RAM, which is used to initialize the data memory, filling it with 0s. Simulators like Modelsim has its own library which contains the HDL file for this IP block. We used the command “-bbox_m altsyncram” to black box the IP block.

```
altsyncram  altsyncram_component (
            .address_a (wadd),
            .address_b (radd),
            .clock0 (clk),
            .data_a (data),
            .wren_a (wren),
            .q_b (sub_wire0),
            .aclr0 (1'b0),
            .aclr1 (1'b0),
            .addressstall_a (1'b0),
            .addressstall_b (1'b0),
            .byteena_a (1'b1),
            .byteena_b (1'b1),
            .clock1 (1'b1),
            .clocken0 (1'b1),
            .clocken1 (1'b1),
            .clocken2 (1'b1),
            .clocken3 (1'b1),
            .data_b ({ramWide{1'b1}}),
            .eccstatus (),
            .q_a (),
            .rden_a (1'b1),
            .rden_b (1'b1),
            .wren_b (1'b0));
defparam
    altsyncram_component.address_aclr_b = "NONE",
    altsyncram_component.address_reg_b = "CLOCK0",
    altsyncram_component.clock_enable_input_a = "BYPASS",
    altsyncram_component.clock_enable_input_b = "BYPASS",
```

1. Feature

MemRead == 1

LB (Load Byte): Funct3 = 3'b000

Loads a byte from memory and sign-extends it to 32 bits.

rd <= {Dataout[7] ? 24'hFFFFFF : 24'b0, Dataout[7:0]};

LH (Load Halfword): Funct3 = 3'b001

Loads a halfword from memory and sign-extends it to 32 bits.

rd <= {Dataout[15] ? 16'hFFFF : 16'b0, Dataout[15:0]};

LW (Load Word): Funct3 = 3'b010

Loads a word from memory.

rd <= Dataout;

LBU (Load Byte Unsigned): Funct3 = 3'b100

Loads a byte from memory and zero-extends it to 32 bits.

rd <= {24'b0, Dataout[7:0]};

LHU (Load Halfword Unsigned): Funct3 = 3'b101

Loads a halfword from memory and zero-extends it to 32 bits.

rd <= {16'b0, Dataout[15:0]}

MemWrite == 1

SB (Store Byte): Funct3 = 3'b000

Stores a byte to memory.

Wr <= (a[1:0]==2'b00) ? 4'b0001 : ((a[1:0]==2'b01) ? 4'b0010 : ((a[1:0]==2'b10) ? 4'b0100 : 4'b1000));

Datain <= (a[1:0]==2'b00) ? {{24{1'b0}}, wd[7:0]} : ((a[1:0]==2'b01) ? {{16{1'b0}}, {wd[7:0], {8{1'b0}}}} : ((a[1:0]==2'b10) ? {{8{1'b0}}, {wd[7:0], {16{1'b0}}}} : {wd[7:0], {24{1'b0}}}));

SH (Store Halfword): Funct3 = 3'b001

Stores a halfword to memory.

Wr <= (a[1:0] == 2'b00 || a[1:0] == 2'b01) ? 4'b0011 : 4'b1100;

Datain <= (a[1:0]==2'b00) || (a[1:0]==2'b01) ? {{16{1'b0}}, wd[15:0]} : {wd[15:0], {16{1'b0}}};
SW (Store Word): Funct3 = 3'b010
Stores a word to memory.
Wr <= 4'b1111;
Datain <= wd;

2. Property and Assertions:

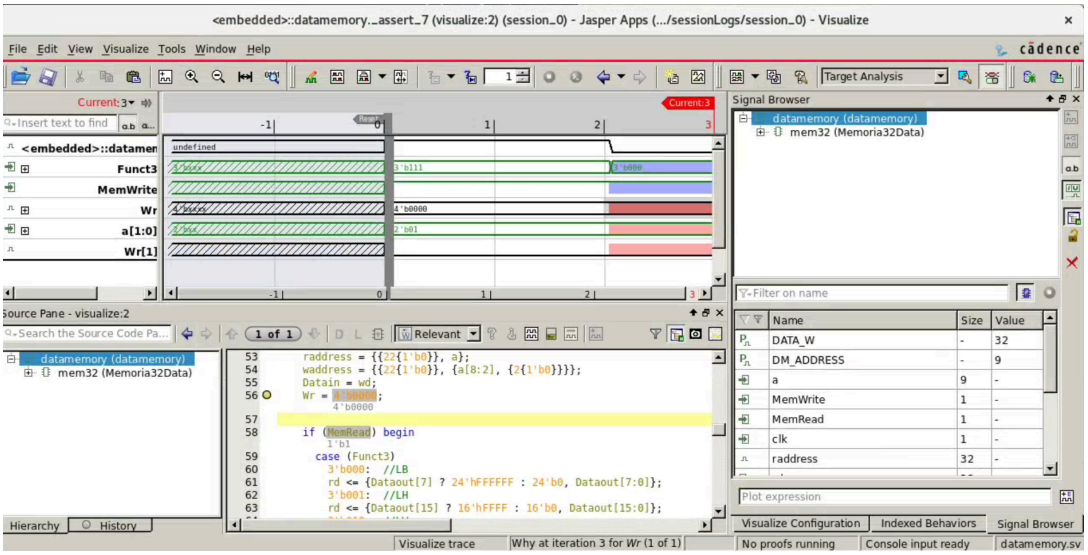
```
always_ff @(*) begin
  // Assertion for MemRead functionality
  if (MemRead) begin
    case (Funct3)
      3'b000: assert (rd == {Dataout[7] ? 24'hFFFFFF : 24'b0, Dataout[7:0]}) else $fatal("LB read failed");
      3'b001: assert (rd == {Dataout[15] ? 16'hFFFF : 16'b0, Dataout[15:0]}) else $fatal("LH read failed");
      3'b010: assert (rd == Dataout) else $fatal("LW read failed");
      3'b100: assert (rd == {24'b0, Dataout[7:0]}) else $fatal("LBU read failed");
      3'b101: assert (rd == {16'b0, Dataout[15:0]}) else $fatal("LHU read failed");
      default: assert (rd == Dataout) else $fatal("Default read failed");
    endcase
  end

  // Assertion for MemWrite functionality
  if (MemWrite) begin
    case (Funct3)
      3'b000: begin  //SB
        assert (Wr == ((a[1:0]==2'b00) ? 4'b0001 : ((a[1:0]==2'b01) ? 4'b0010 : ((a[1:0]==2'b10) ? 4'b0100 : 4'b1000)))) else $fatal("SB write failed");
        assert (Datain == ((a[1:0]==2'b00) ? {{24{1'b0}}, wd[7:0]} : ((a[1:0]==2'b01) ? {{16{1'b0}}, {wd[7:0], {8{1'b0}}}} : ((a[1:0]==2'b10) ? {{8{1'b0}}, {wd[7:0], {16{1'b0}}}} : {wd[7:0], {24{1'b0}}})))) else $fatal("SB data write failed");
      end
      3'b001: begin  //SH
        assert (Wr == ((a[1:0] == 2'b00 || a[1:0] == 2'b01) ? 4'b0011 : 4'b1100)) else $fatal("SH write failed");
        assert (Datain == ((a[1:0]==2'b00) || (a[1:0]==2'b01) ? {{16{1'b0}}, wd[15:0]} : {wd[15:0], {16{1'b0}}})) else $fatal("SH data write failed");
      end
      default: begin  //SW
        assert (Wr == 4'b1111) else $fatal("SW write failed");
        assert (Datain == wd) else $fatal("SW data write failed");
      end
    endcase
  end
end
```

3. Results

	Type	Name	Engine	Bound	Trace
✔	Assert	datamemory._assert_1	N (1)	Infinite	0
✔	Cover (related)	datamemory._assert_1:precondition1	N	1	1
✔	Assert	datamemory._assert_2	N (1)	Infinite	0
✔	Cover (related)	datamemory._assert_2:precondition1	N	1	1
✔	Assert	datamemory._assert_3	N (1)	Infinite	0
✔	Cover (related)	datamemory._assert_3:precondition1	N	1	1
✔	Assert	datamemory._assert_4	N (1)	Infinite	0
✔	Cover (related)	datamemory._assert_4:precondition1	N	1	1
✔	Assert	datamemory._assert_5	Hp (1)	Infinite	0
✔	Cover (related)	datamemory._assert_5:precondition1	Hp	1	1
✔	Assert	datamemory._assert_6	Hp (1)	Infinite	0
✔	Cover (related)	datamemory._assert_6:precondition1	Hp	1	1
✘	Assert	datamemory._assert_7	Hp	1	1
✔	Cover (related)	datamemory._assert_7:precondition1	Hp	1	1
✘	Assert	datamemory._assert_8	Hp	1	1
✔	Cover (related)	datamemory._assert_8:precondition1	Hp	1	1
✘	Assert	datamemory._assert_9	Hp	1	1
✔	Cover (related)	datamemory._assert_9:precondition1	Hp	1	1

Expression _assert_7
Status ✘ CEX Bound 1
Time 0.1 Engine Hp
Filename datamemory.sv

```
108    assert (Wr == ((a[1:0]==2'b00) ? 4'b0001 : ((a[1:0]==
```



The error happened on the wr signal. We extended the clk to 3 cycles, but the wr signal remains 0. We believe the reason why the wr signal is 0 is that the wr signal has both blocking and non-blocking assignments in the always block. Regarding the question of why bbox has no effect on this result, we think that bbox only affects the initialization process of data, which may have a negative impact on memory read. However, for memory write, data comes from datain and is unrelated to the initialization process.

4. How to run the JG for this component.
under the folder “datamemory”, type the following command in the terminal:
./run.sh

```
always_ff @(*) begin
  raddress = {{22{1'b0}}, a};
  waddress = {{22{1'b0}}, {a[8:2], {2{1'b0}}}};
  Datain = wd;
  Wr = 4'b0000;

  if (MemRead) begin
    case (Funct3)
      3'b000:  //LB
      rd <= {Dataout[7] ? 24'hFFFFFF : 24'b0, Dataout[7:0]};
      3'b001:  //LH
      rd <= {Dataout[15] ? 16'hFFFF : 16'b0, Dataout[15:0]};
      3'b010:  //LW
      rd <= Dataout;
      3'b100:  //LBU
      rd <= {24'b0, Dataout[7:0]};
      3'b101:  //LHU
      rd <= {16'b0, Dataout[15:0]};
      default: rd <= Dataout;
    endcase
  end else if (MemWrite) begin
    case (Funct3)
      3'b000: begin  //SB
        Wr <= (a[1:0]==2'b00) ? 4'b0001 : ((a[1:0]==2'b01) ? 4'b0010 : ((a[1:0]==2'b10) ? 4'b0100 : 4'b1000))
        Datain <= (a[1:0]==2'b00) ? {{24{1'b0}}, wd[7:0]} : ((a[1:0]==2'b01) ? {{16{1'b0}}, {wd[7:0], {8{1'b0
      end
      3'b001: begin  //SH
        Wr <= (a[1:0] == 2'b00 || a[1:0] == 2'b01) ? 4'b0011 : 4'b1100;
        Datain <= (a[1:0]==2'b00) || (a[1:0]==2'b01) ? {{16{1'b0}}, wd[15:0]} : {wd[15:0], {16{1'b0}}};
      end
      default: begin  //SW
        Wr <= 4'b1111;
        Datain <= wd;
      end
    endcase
  end
end
```

Conclusion:

The open-source RISC-V processor design includes its own verification process, as shown in the repository. However, through formal verification using tools like JasperGold, our analysis reveals that the design fails to meet certain formal verification requirements. Specifically:

1. Verification Challenges Identified:
   The wr signal issue demonstrates improper handling of blocking and non-blocking assignments in the design, causing unexpected behavior during memory write operations.
   The bbox (black-boxing) approach used in the verification does not mitigate these issues effectively, as it primarily affects the initialization of data but has no influence on memory write paths where datain is directly used.
2. Insights Gained:
   The design's formal verification revealed stricter standards compared to simulation-based validation, uncovering errors that were not apparent during the original design verification.

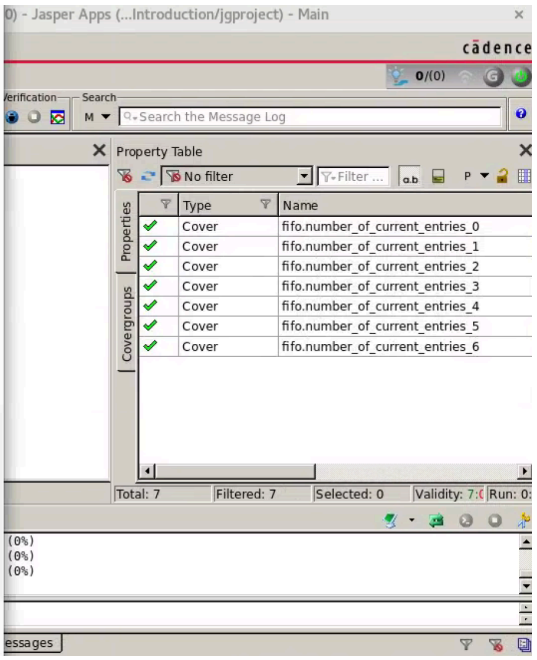
This highlights the importance of incorporating exhaustive property-based formal verification methodologies to uncover edge cases and ensure robust hardware designs.

Appendix: Two FIFO Problem

Task 1.

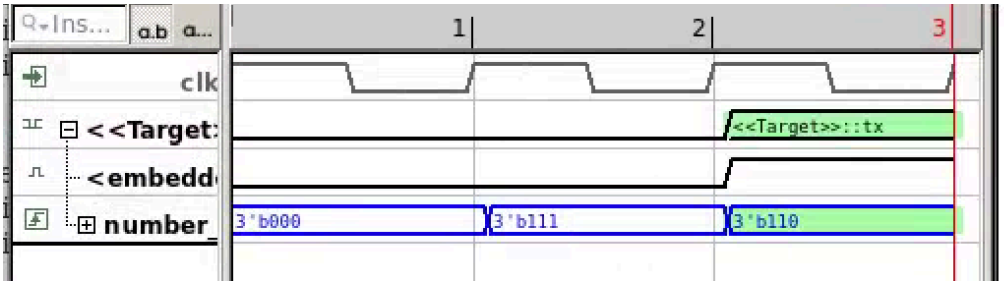
A:

```
76 // Cover property for number_of_current_entries = 0
77 number_of_current_entries_0:cover property (@(posedge clk)
   number_of_current_entries == 0)
78     $display("Cover hit: FIFO is empty (number_of_current_entries =
   0)");
79
80 // Cover property for number_of_current_entries = 1
81 number_of_current_entries_1:cover property (@(posedge clk)
   number_of_current_entries == 1)
82     $display("Cover hit: FIFO has 1 entry (number_of_current_entries =
   1)");
83
84 // Cover property for number_of_current_entries = 2
85 number_of_current_entries_2:cover property (number_of_current_entries
   == 2)
86     $display("Cover hit: FIFO has 2 entries (number_of_current_entries
   = 2)");
87
88 // Cover property for number_of_current_entries = 3
89 number_of_current_entries_3:cover property (number_of_current_entries
   == 3)
90     $display("Cover hit: FIFO has 3 entries (number_of_current_entries
   = 3)");
91
92 // Cover property for number_of_current_entries = 4 (full)
93 number_of_current_entries_4:cover property (number_of_current_entries
   == 4)
94     $display("Cover hit: FIFO is full (number_of_current_entries =
   4)");
95
96 // Cover property for number_of_current_entries = 5 (overflow
   condition)
97 number_of_current_entries_5:cover property (number_of_current_entries
   == 5)
98     $display("Cover hit: FIFO overflow condition
   (number_of_current_entries = 5)");
```



B: From the design point of view, the number of current entries should not increase to 5 and 6, for a 4 deep FIFO. The testing result shows that all of the number_of_current_entries = 0, 1, 2, 3, 4, 5, 6 are coverable.

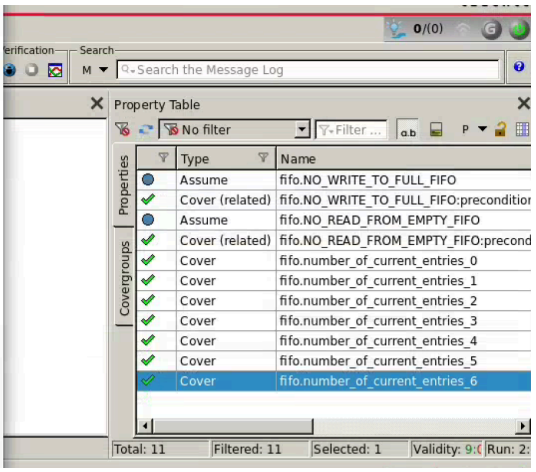
C:



From the waveform which analyzes if number_of_current_entries==6, it can be seen that the 3 bits number_of_current_entries are possible to cover numbers of 0-7. Also, the inputs overdraw the entries from the FIFO, i.e. reading from empty FIFO. Some assumptions are required to make this work.

D:

```
75 NO_WRITE_TO_FULL_FIFO:assume property (
76         out_is_full |-> !in_write_ctrl); // Do not write if the fifo
   is full
77 NO_READ_FROM_EMPTY_FIFO:assume property (
78         out_is_empty |-> !in_read_ctrl); // Do not read if the fifo is
   empty
79
80 // Cover property for number_of_current_entries = 0
81 number_of_current_entries_0:cover property (@(posedge clk)
   number_of_current_entries == 0)
82     $display("Cover hit: FIFO is empty (number_of_current_entries =
   0)");
83
84 // Cover property for number_of_current_entries = 1
85 number_of_current_entries_1:cover property (@(posedge clk)
   number_of_current_entries == 1)
86     $display("Cover hit: FIFO has 1 entry (number_of_current_entries =
   1)");
87
88 // Cover property for number_of_current_entries = 2
89 number_of_current_entries_2:cover property (number_of_current_entries
   == 2)
90     $display("Cover hit: FIFO has 2 entries (number_of_current_entries
   = 2)");
91
```



The two assumptions are added to the property check (line 75- 78), so that the environment does not read from an empty FIFO and does not write to a full FIFO.

E: the running result is shown on the right side of the picture above.

F: the number of entries of 5 and 6 are still possible, which is unexpected.

From the waveform, it can be seen that the out_is_empty signal switches to 0 unexpectedly while the FIFO is empty, indicating there is a bug on the out_is_empty signal.

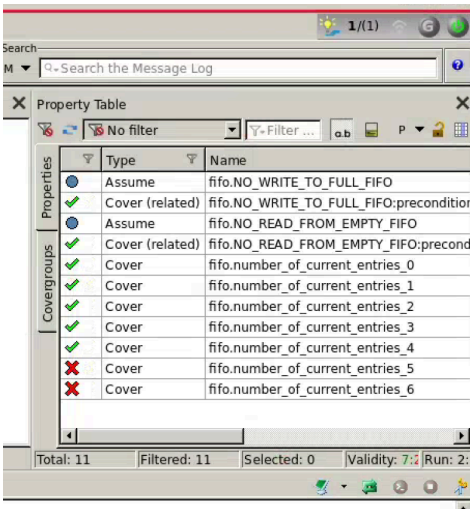
G:

```
53
54 always_ff @(posedge clk) begin
55    if (rst) begin
56       number_of_current_entries <= 0;
57       out_is_empty <= 1;
58       out_is_full <= 0;
59    end
60    else if (in_read_ctrl & ~in_write_ctrl ) begin
61       number_of_current_entries <= number_of_current_entries - 1'b1;
62       out_is_full <= 0;
63       out_is_empty <= (number_of_current_entries == 1'b1);
64    end
65    else if (~in_read_ctrl & in_write_ctrl) begin
66       number_of_current_entries <= number_of_current_entries + 1'b1;
67       out_is_empty <= 0;
68       out_is_full <= (number_of_current_entries == (ENTRIES-1'b1));
69    end
70     else if (~in_read_ctrl & ~in_write_ctrl) begin
71       out_is_empty <= 0;
72       out_is_full <= 0;
73    end
74 end
75
76 endmodule
```

In line 68-69, the out_is_empty and out_is_full should not set to 0. They should keep their value when there is no reading and writing.

H: these bugs are fixed as shown in the following screenshot:

```
54 always_ff @(posedge clk) begin
55    if (rst) begin
56       number_of_current_entries <= 0;
57       out_is_empty <= 1;
58       out_is_full <= 0;
59    end
60    else if (in_read_ctrl & ~in_write_ctrl ) begin
61       number_of_current_entries <= number_of_current_entries - 1'b1;
62       out_is_full <= 0;
63       out_is_empty <= (number_of_current_entries == 1'b1);
64    end
65    else if (~in_read_ctrl & in_write_ctrl) begin
66       number_of_current_entries <= number_of_current_entries + 1'b1;
67       out_is_empty <= 0;
68       out_is_full <= (number_of_current_entries == (ENTRIES-1'b1));
69    end
70     else if (~in_read_ctrl & ~in_write_ctrl) begin
71       out_is_empty <= out_is_empty;
72       out_is_full <= out_is_full;
73    end
74 end
75
76 endmodule
77
```

I: 

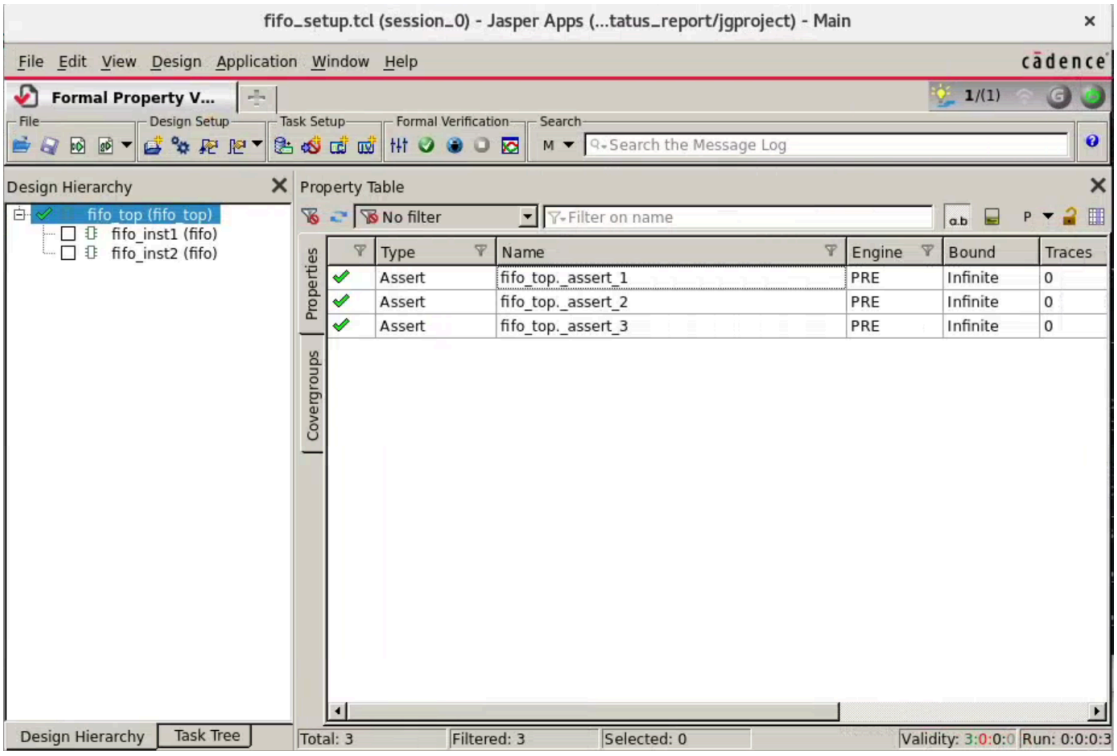
Now the results match the expectation, where there is only 0, 1, 2, 3, 4 are coverable in the number_of_current_entries variable.

Task 2

A: By instantiating the same design twice, we get this:

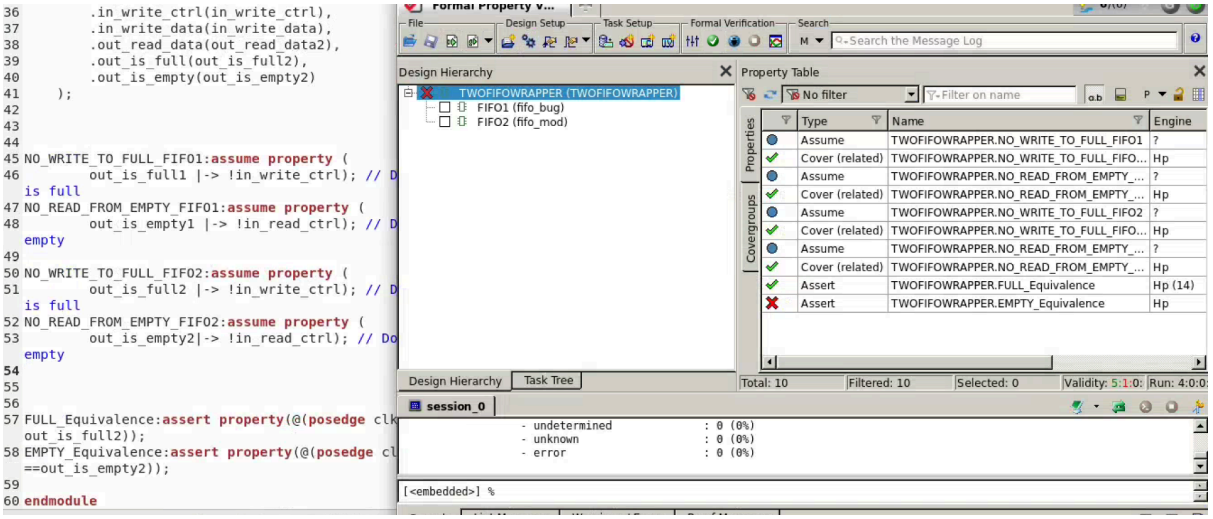
```
fifo fifo_inst1 (
    .clk(clk),
    .rst(rst),
    .in_read_ctrl(in_read_ctrl),
    .in_write_ctrl(in_write_ctrl),
    .in_write_data(in_write_data),
    .out_read_data(out_read_data1),
    .out_is_full(out_is_full1),
    .out_is_empty(out_is_empty1)
);

fifo fifo_inst2 (
    .clk(clk),
    .rst(rst),
    .in_read_ctrl(in_read_ctrl),
    .in_write_ctrl(in_write_ctrl),
    .in_write_data(in_write_data),
    .out_read_data(out_read_data2),
    .out_is_full(out_is_full2),
    .out_is_empty(out_is_empty2)
);

assert property (@(posedge clk) disable iff (rst) (out_read_data1 == out_read_data2));
assert property (@(posedge clk) disable iff (rst) (out_is_full1 == out_is_full2));
assert property (@(posedge clk) disable iff (rst) (out_is_empty1 == out_is_empty2));
```

Three assertions have been proved.

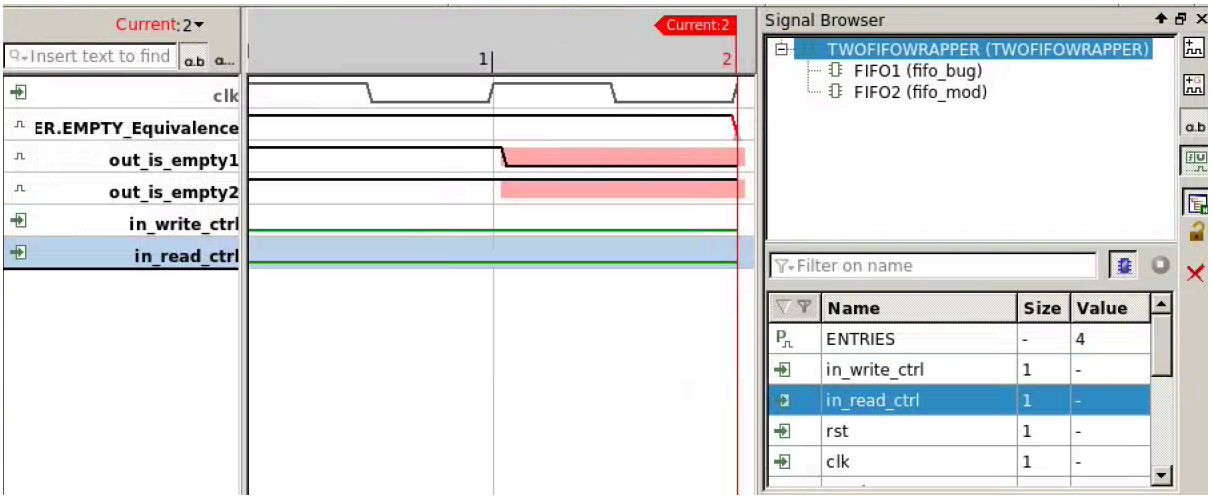


B: introducing a bug on line 69: the out_is_empty will be set to 0 wrongly.

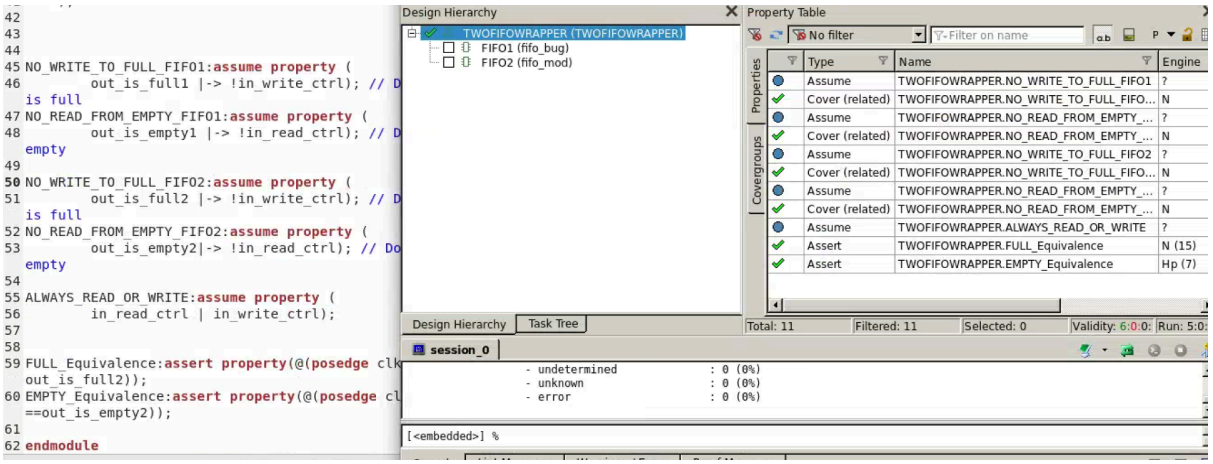
```
68     else if (~in_read_ctrl & ~in_write_ctrl) begin
69       out_is_empty <= 0;
70       out_is_full <= out_is_full;
71     end
```



Only investigating the out_is_empty assert property, the bug is being found as an assertion failed. The waveform below shows the bug is triggered when the in_read_ctrl and in_write_ctrl are both 0.



C: overconstraining the input is possible to “Prove” the equivalence.



For example, the above picture shows the additional assumption which says there is always a read or write command, which means it is impossible to have both in_read_ctrl and in_write_ctrl == 0. This overconstraint makes the jasper gold pass the assertion, and proves that it is possible to overconstrain the inputs in part B to “prove” that the two designs are equivalent.